November 1988

UILU-ENG-88-2257
ACT-103

COORDINATED SCIENCE LABORATORY
College of Engineering
Applied Computation Theory

FAST AND PROCESSOR-EFFICIENT PARALLEL ALGORITHMS FOR REDUCIBLE FLOW GRAPHS

Vijaya Ramachandran

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS None
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) UILU-ENG-88-2257 (ACT #103)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Coordinated Science Lab University of Illinois 6b. OFFICE SYMBOL (If applicable) N/A	7a. NAME OF MONITORING ORGANIZATION Office of Naval Research
6c. ADDRESS (City, State, and ZIP Code) 1101 W. Springfield Ave. Urbana, IL 61801	7b. ADDRESS (City, State, and ZIP Code) Arlington, VA 22217
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Joint Services Electronics Program 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER N00014-84-C-0149
8c. ADDRESS (City, State, and ZIP Code) Arlington, VA 22217	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)

Fast and Processor-Efficient Parallel Algorithms for Reducible Flow Graphs

12. PERSONAL AUTHOR(S)

Ramachandran, Vijaya

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Technical	FROM _____ TO _____	November, 1988	26

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	algorithms, directed graphs, flow graphs, parallel computation, PRAM model.

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This document presents ~~We present~~ parallel NC algorithms for recognizing reducible flow graphs, and for finding dominators, minimum feedback vertex sets, and a depth first search numbering in an rfg. All of these algorithms run in polylog parallel time using $M(n)$ processors, where $M(n)$ is the number of processors needed to multiply two $n \times n$ matrices in polylog time; this is the best processor bound currently known for polylog-time parallel algorithms for directed graphs.

It is shown ~~We show~~ that finding a minimum feedback vertex set in vertex-weighted rfg's or finding a minimum feedback arc set in arc-weighted rfg's is P-complete. For arc or vertex weights in unary, we present RNC algorithms for these problems and show that these problems are in NC if and only if the problem of finding a maximum matching is in NC.

Keywords: parallel Computations. (KR)

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code) / 22c. OFFICE SYMBOL

DD Form 1473, JUN 86 Previous editions are obsolete. SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED

FAST AND PROCESSOR-EFFICIENT PARALLEL ALGORITHMS FOR REDUCIBLE FLOW GRAPHS†

Vijaya Ramachandran

Coordinated Science Laboratory

University of Illinois, Urbana, IL 61801

Tech. Report ACT-103

November 1988

ABSTRACT

We present parallel NC algorithms for recognizing reducible flow graphs, and for finding dominators, minimum feedback vertex sets, and a depth first search numbering in an rfg. All of these algorithms run in polylog parallel time using $M(n)$ processors, where $M(n)$ is the number of processors needed to multiply two $n \times n$ matrices in polylog time; this is the best processor bound currently known for polylog-time parallel algorithms for directed graphs.

We show that finding a minimum feedback vertex set in vertex-weighted rfg's or finding a minimum feedback arc set in arc-weighted rfg's is P-complete. For arc or vertex weights in unary, we present RNC algorithms for these problems and show that these problems are in NC if and only if the problem of finding a maximum matching is in NC.

†This work was funded by the Joint Services Electronics Program under grant N00014-84-C-0149.

1. Introduction

Reducible flow graphs (rfg's) are graphs that model the control structure of computer programs. They are used extensively in problems on code optimization and global data flow analysis. Several linear time sequential algorithms for these graphs are known, including algorithms for recognizing rfg's [Ta1, GaTa], for finding *dominators* [Ha] and for finding a *minimum feedback vertex set (FVS)* [Sh]. The basis for all of these fast sequential algorithms is a depth first search on the input graph. Recently, we have developed polynomial-time algorithms for finding a minimum *weight* FVS in vertex-weighted rfg's and a minimum *feedback arc set (FAS)* in arc-weighted or unweighted rfg's [Ra1]. These algorithms make extensive use of algorithms for network flow [FoFu, La, PaSt, Ta2, GoTa]. It is also known that the sequential complexity of these latter problems is at least that of finding a minimum cut in a flow network [Ra1, Ra2].

In this paper we give parallel NC algorithms for recognizing rfg's, for finding dominators, and for finding a minimum FVS in an unweighted rfg. We note that the problem of finding a minimum FVS in *cyclically reducible graphs*, a class closely related to rfg's, is reported to be P-complete in [BoDAPe]. We also give an NC algorithm for finding a depth first search (DFS) numbering for an rfg; however, none of our other parallel algorithms make use of this DFS numbering. The processor bound for all of these NC algorithms is the number of processors need by an NC algorithm to multiply two $n \times n$ matrices. This bound is good with respect to current NC algorithms for directed graphs, since most of these algorithms require this number of processors.

We show that if arbitrary weights are allowed, the weighted FAS and FVS problems on rfg's are both P-complete. Hence fast parallel algorithms for these problems appear unlikely to exist. For the case when the weights are in unary, we present an RNC algorithm for the FAS problem on rfg's. We also give NC reductions between the weighted FAS problem, the unweighted FAS problem, the weighted FVS problem and the problem of finding a minimum cut in a flow network (when weights and capacities are in unary). Thus if any one of these problems is in NC, then all of them would be in NC. In particular, an NC algorithm for the maximum

matching problem would give NC algorithms for these three problems on rfg's, and an NC algorithm for any one of these three problems would, in turn, give an NC algorithm for maximum matching.

A preliminary version of this paper appeared in [Ra3]. Some of the NC algorithms in the present paper use a smaller number of processors than the corresponding ones in [Ra3].

This paper is organized as follows. In section 2 we provide definitions. In section 3, we present our parallel algorithms for preprocessing an rfg. In section 4 we present a parallel algorithm for finding a minimum FVS in an unweighted rfg. Finally in section 5 we give an RNC algorithm for finding a minimum FAS in an unweighted rfg, and present P-completeness results for the weighted FAS and FVS problems on rfg's.

2. Definitions

2.1. Model of Parallel Computation

The parallel model of computation that we will be using is the *PRAM* model, which consists of several independent sequential processors, each with its own private memory, communicating with one another through a global memory. In one unit of time, each processor can read one global or local memory, execute a single RAM operation, and write into one global or local memory location.

PRAMs are classified according to restrictions on global memory access. An EREW PRAM is a PRAM for which simultaneous access to any memory location by different processors is forbidden for both reading and writing. In a CREW PRAM simultaneous reads are allowed but no simultaneous writes. A CRCW PRAM allows simultaneous reads and writes. In this case we have to specify how to resolve write conflicts. We will use the COMMON model in which all processors participating in a concurrent write must write the same value. Of the three PRAM models we have listed, the EREW model is the most restrictive, and the COMMON CRCW model is the most powerful. It is not difficult to see that any algorithm for the COMMON CRCW PRAM that runs in parallel time T using P processors can be simulated by an EREW PRAM (and hence by a

CREW PRAM) in parallel time $T\log P$ using the same number of processors, P.

Define $polylog(n)=\bigcup_{k\geq 0} O(\log^k n)$. The class NC is the class of problems solvable in $polylog(n)$ parallel time with a number of processors polynomial in n, where n is the size of the input. This class is generally accepted to characterize the class of problems that can be solved feasibly in parallel.

The class P is the class of problems solvable by a sequential algorithm running in polynomial time. A problem is *P-complete* if every problem in P can be reduced to it in logspace. A P-complete problem is in NC if and only if NC=P. Since it is widely conjectured that NC is a proper subset of P, showing a problem to be P-complete is strong evidence that the problem is not in NC.

For problems in NC, we would like to develop algorithms that run in polylog parallel time and also use a small number of processors. For undirected graphs there are algorithms known for several problems that run in polylog time using a linear number of processors (or less) on a PRAM; these problems include graph connectivity, biconnectivity and triconnectivity, s–t numbering, planarity, etc. For directed graphs unfortunately, such efficient parallel algorithms are not known mainly due to the *transitive closure bottleneck*. The best parallel method known at present to test reachability from one vertex to another in a directed graph is to find the transitive closure of the adjacency matrix of the graph. To compute this in polylog time requires n^α processors (to within a polylog factor), where α is the *matrix multiplication exponent*, which is currently 2.375 (but for practical computations should be taken as 3). Thus, since rfg's are directed graphs, all of the algorithms we present in this paper are affected by the transitive closure bottleneck.

The algorithms we develop in this paper make use of some well-known basic parallel algorithms as subroutines. We conclude this section with a brief review of these algorithms. For more on the PRAM model and PRAM algorithms see [KarRa].

1. Boolean matrix multiplication and transitive closure: The standard matrix multiplication algorithm can be parallelized to give a constant time algorithm using n^3 processors on a COMMON

PRAM to find the product of two $n \times n$ Boolean matrices. Since $(I+B)^m$, for $m \geq n$, gives the transitive closure B^* of an $n \times n$ Boolean matrix B, B^* can be computed using $\log n$ stages of Boolean matrix multiplication by repeated squaring and thus in $O(\log n)$ time on a COMMON PRAM with n^3 processors. The more sophisticated matrix multiplication algorithms (that work for matrices over a ring, and can be adapted to Boolean matrix multiplication) lend themselves to parallelization on an EREW or CREW PRAM. Thus multiplication of two $n \times n$ Boolean matrices can be done in $O(\log n)$ time with $M(n)=n^\alpha$ processors on an EREW PRAM, and hence B^* can be obtained in $O(\log^2 n)$ time using $M(n)$ processors on an EREW PRAM.

In many of the algorithms we present, the processor count is dominated by the number of processors needed to multiply two $n \times n$ Boolean matrices. The steps that do not require matrix multiplication typically need $O(n^2)$ processors. In such cases, we will state our processor-time bound as parallel time $t(n)$ using $Q(n)$ processors. This will imply that the algorithm runs in time $t(n)$ with $O(n^3)$ processors on a COMMON PRAM, and it runs in time $O(t(n) \cdot \log n)$ with $O(n^\alpha)$ processors on an EREW or CREW PRAM. Any future improvement in the processor count for matrix multiplication on any of these types of PRAM will cause a corresponding improvement in the processor bound for the algorithm, since the remaining steps need $O(n^2)$ processors.

2. *Prefix sums:* Let + be an associative operation over a domain D. Given an ordered list $\langle x_1, \cdots, x_n \rangle$ of n elements from D, the prefix problem is to compute the $n-1$ prefix sums $S_i = \sum_{j=1}^{i} x_j, i=1, \cdots, n$. This problem has several applications. For example, consider the problem of compacting a sparse array, i.e., we are given an array of n elements, many of which are zero, and we wish to generate a new array containing the nonzero elements in their original order. We can compute the position of each nonzero element in the new array by assigning value 1 to the nonzero elements, and computing prefix sums with + operating as regular addition.

The n element prefix sums problem can be computed in $O(\log n)$ time using $n/\log n$ processors on a EREW PRAM, assuming unit time for a single + operation.

3. List Ranking: This is a generalization of prefix sums, in which the ordered list is given in the form of a linked list rather than an array. List ranking on n elements can be computed by a simple algorithm in $O(\log n)$ time using n processors on an EREW PRAM; more elaborate algorithms for the problem run in $O(\log n)$ time using $n/\log n$ processors.

4. Tree contraction: Tree contraction is a method of evaluating tree functions efficiently in parallel. The method transforms the input tree using two operations Rake and Compress. The operation Rake removes leaves from the tree. The operation Compress halves the lengths of chains in the tree (a *chain* is a sequence of vertices with exactly one incoming and outgoing arc in the tree) by removing alternate nodes in the chain and linking each remaining node to the parent of its parent in the original tree. The Contract operation is one application of Rake followed by one application of Compress. It can be shown [MiRe] that $O(\log n)$ applications of the Contract operation to an n node tree are sufficient to transform the tree into a single vertex. This contraction can be done in $O(\log n)$ time with n processors on a CREW PRAM. More elaborate algorithms for this problem run in $O(\log n)$ time with $n/\log n$ processors on an EREW PRAM.

Some of the algorithms we will present in this paper will use a modified tree contraction method, which we describe at the end of section 3.

2.2. Graph-theoretic Definitions

A *directed graph* $G=(V,A)$ consists of a finite set of vertices (or nodes) V and a set of arcs A which is a subset of $V \times V$. An arc $a=(v_1,v_2)$ is an *incoming* arc to v_2 and an *outgoing* arc from v_1. Vertex v_1 is a *predecessor* of v_2 and v_2 is a *successor* of v_1; v_1 is the *tail* of a and v_2 is its *head*. Given a directed graph $G=(V,A)$ and a set of arcs C, we will sometimes use the notation $C \cap G$ to denote the set $C \cap A$. An *arc-weighted (vertex-weighted) directed graph* is a directed graph with a real value on each arc (vertex).

A *directed path* p in G from vertex u to vertex v is a sequence of arcs $a_1, \cdots, a_r$ in A such that $a_i=(w_i,w_{i+1}), i=1, \cdots, r$ with $w_1=u$ and $w_{r+1}=v$. The path p *passes through* each $w_i, i=1, \cdots, r+1$. A directed path p from u to v is a *cycle* if $u=v$. A DAG is a directed acyclic

graph, i.e., a directed graph with no cycle. A *rooted directed graph* or a *flow graph* $G=(V,A,r)$ is a directed graph with a distinguished vertex r such that there is a directed path in G from r to every vertex v in $V-\{r\}$.

A *rooted tree* is a flow graph $T=(V,A,r)$ in which every vertex in $V-\{r\}$ has exactly one incoming arc. If (u,v) is the unique incoming arc to v then u is the *parent* of v, and v is a *child* of u. A *leaf* is a vertex in a tree with no outgoing arc. The *height* of a vertex v in a tree is the length of a longest path from v to a leaf. The *height of a tree* is the height of its root. A *forest* is a collection of trees.

Let $G=(V,A,r)$ be a rooted DAG. A vertex u is a descendant of vertex v if either $u=v$ or there is a directed path from v to u in G. The vertex u is a *proper descendant* of v if $u\neq v$ and u is a descendant of v.

Let $G=(V,A)$ be an arc-weighted directed graph. A set $F\subseteq A$ is a *feedback arc set (FAS)* for G if $G'=(V,A-F)$ is acyclic. The set F is a *minimum* FAS if the sum of the weights of arcs in F is minimum. Analogous definitions hold for a *feedback vertex set.*

Let $G=(V,A)$ be a directed graph, and let $V'\subseteq V$. The *subgraph of G induced by V'* is the graph $G_s(V')=(V',A')$, where $A'=A\cap V'XV'$. The graph $G-V'$ is the subgraph of G induced on $V-V'$.

A *reducible (flow) graph* (or *rfg)* is a rooted directed graph for which the rooted depth first search DAG [Ta2] is unique. Thus, the arcs in a reducible graph can be partitioned in a unique way into two sets as the *DAG* or *forward* arcs and the *back* arcs.

An alternate definition of a reducible graph (due to [HeU1]) is stated below.

Definition 2.1 [HeU1] Let $G=(V,A,r)$ be a flow graph. We define two transformations on G:

Transformation T_1: Given an arc $a=(v,v)$ in A remove a from A.

Transformation T_2: Let v_2 be a vertex in $V-\{r\}$ and let it have a single incoming arc $a=(v_1,v_2)$. T_2 replaces v_1,v_2 and a by a single vertex v. Predecessors of v_1 become predecessors of v. Successors of v_1 and v_2 become successors of v. There is an arc (v,v) if and

only if there was formerly an arc (v_2,v_1) or (v_1,v_1).

G is a *reducible flow graph (rfg)* if repeated applications of T_1 and T_2 (in any order) reduce G to a single vertex.

Let $G=(V,A,r)$ be a reducible graph and let $b=(u,v)$ be a back arc in G. Then b *spans* vertex w (or w is in the *span* of b) if there exists a path from v to u in the DAG of G that passes through w. Given two vertices $u,v \in V$, vertex u *dominates* vertex v if every path from r to v passes through u (note that u dominates itself).

It is well-known [AhUl] that the dominator relation can be represented in the form of a tree rooted at r, the root of the flow graph G. This tree is called the *dominator tree* T of G. The descendents of a vertex v in T are the vertices dominated by v in G. A vertex v' is *immediately dominated* by v if it is a child of v in T.

Given a set $V' \subseteq V$ the *dominator forest* $F_{V'}$ *for* V' represents the dominator relation restricted to the set V'. Let $V_h=\{v \in V \mid v$ is the head of a back arc in $G\}$. We assume that r is the head of a back arc in G. Hence it is easy to see that F_{V_h} is a tree; we call it the *head dominator tree of* G and denote it by T_h. This tree can be constructed from T by applying transformation T_2 of Definition 2.1 to each vertex v in T that is not a head of a back arc in G.

3. Parallel Algorithms for Preprocessing RFG's

In this section we present NC algorithms to test if a rooted directed graph is an rfg, to construct the head dominator tree for an rfg, and to find a DFS tree in an rfg; we also introduce a modified tree contraction method in this section.

3.1 Testing flow graph reducibility:

Input: $G=(V,A,r)$ with adjacency matrix B.

1. Test if G is a flow graph, i.e., test if every vertex in $V-\{r\}$ is reachable from r.

   Form B^*, the transitive closure of B and check if every nondiagonal element in row r has a 1.

2. Construct a tree T rooted at r using the algorithm in [GaMi] to find a directed breadth-first search tree. Mark all arcs in T as forward (f) arcs.

3. For each arc (u,v) in $G-T$, mark (u,v) as b if v is an ancestor of u, and as f otherwise.

4. Delete all arcs marked b and check if resulting graph G' is acyclic. (If G is an rfg then G' must be acyclic.)

Form transitive closure of the adjacency matrix B' of G' and check that for every $i<j\leq n$, one of the two entries in position (i,j) and position (j,i) in B'^* is zero.

5. Compute dominators in G' using the algorithm in [PaGoRa].

6. For each arc (u,v) marked b in G, check if v dominates u. G is a rfg if and only if v dominates u for all arcs (u,v) marked b.

Lemma 3.1 Algorithm 3.1 correctly determines if the input graph is an rfg.

Proof If G is an rfg, then its arcs can be partitioned in a unique way as forward and back arcs, and for any back arc $b=(u,v)$, vertex v must dominate vertex u [HeUl]. Hence the tree T found in step 1 must contain only forward arcs of G. Further, if an arc (u,v) not in T is a back arc of G, then v must be an ancestor of u in T. Further, consider any arc (u,v), with v an ancestor of u in T. Then arc (u,v) completes a cycle in G, consisting of itself, followed by the path in T from v to u. One of these arcs must be a back arc. Since all of the arcs in T are forward arcs, it follows that (u,v) must be a back arc. Hence steps 2 and 3 of Algorithm 3.1 correctly identify the forward and back arcs of G if G is an rfg.

A flow graph is an rfg if and only if its set of arcs can be partitioned into two sets E_1 and E_2 such that E_1 forms an acyclic subgraph D of G, and for each $a=(u,v)$ in E_2, v dominates u in D [HeUl]. Thus all of the tests in steps 4, 5 and 6 are satisfied if and only if G is an rfg.[]

Steps 1, 2, 4 and 5 take $O(\log n)$ time using $Q(n)$ processors. Steps 3 and 6 can be implemented in $O(\log n)$ time using a linear number of processors using tree contraction. Hence the complexity of this algorithm is $O(\log n)$ parallel time using $Q(n)$ processors.

3.2 Forming T_h, the head dominator tree for G :

1. Use algorithm 3.1 to construct DAG G'.

2. Use the algorithm in [PaGoRa] to compute the dominator tree T for DAG G'.

3. Use tree contraction to extract the head dominator tree T_h from T.

Steps 1 and 2 take $O(\log n)$ time with $Q(n)$ processors, and step 3 takes $O(\log n)$ time with n processors. Hence step 1 dominates the complexity of this algorithm.

3.3. NC algorithm for finding a DFS numbering for an rfg $G=(V,A,r)$

1. Use algorithm 3.1 to construct the DAG G'.

2. Find a DFS tree in DAG G' as follows.

i) Identify a vertex v with more than $n/2$ descendents for which every child has at most $n/2$ descendents:

Find the transitive closure B'^* of B'. Determine the number of descendents of each vertex as the sum of the nondiagonal entries in its row in B'^*. Each arc in G' compares the number of descendents of its head with the number of descendents of its tail, and *marks* its tail if it is not the case that the head has at most $n/2$ descendents and the tail has more than $n/2$ descendents. The (unique) unmarked vertex is v.

ii) Find a path P from root r to v:

Find a directed spanning tree for G' by making each vertex with an incoming arc choose one such arc as its tree arc. Form P as the path in this tree from r to v.

iii) Associate each descendent v' of v with the largest numbered child of v (numbering according to some fixed order) from which it is reachable; associate each vertex v' not reachable from v with the lowest vertex in path P from which it is reachable:

Use list ranking to number the vertices on P in increasing order from the root, followed by the children of v in some fixed order. Replace all nonzero entries in the columns of B'^*

corresponding to these nodes by their new number. For each row, find the maximum numbered entry in that row, and identify it as the vertex with which the row vertex is to be associated.

iv) Recursively solve problem in subdags rooted at the newly numbered vertices, together with their descendants as computed in step iii.

Lemma 3.2 Let $G=(V,A,r)$ be a DAG, with $|V|=n$. There exists a unique vertex $u \in V$ with more than $n/2$ descendants for which every child has at most $n/2$ descendants.

Proof Straightforward, and is omitted.[]

Lemma 3.3 Algorithm 3.3 correctly finds a DFS tree in an rfg.

Proof We observe that the algorithm constructs a DFS tree consisting of the initial path P to v, followed by a DFS on the vertices reachable from the children of the largest numbered child of v, followed by vertices reachable from the second largest numbered child of v (but not reachable from the largest child of v), ... , followed by vertices reachable from the smallest numbered child of v (but not reachable from larger numbered children of v), followed by vertices reachable from nodes on $P-\{v\}$ in reverse order of their occurrence on P. It is not difficult to see that this is a valid depth first search.[]

Step 2i takes $O(\log n)$ time using $Q(n)$ processors. Step 2ii is very efficient: it takes constant time using a linear number of processors on an EREW PRAM. Step 2iii takes $O(\log n)$ time using n^2 processors on an EREW PRAM. Finally the recursive steps take $\log n$ stages since each new subproblem is at most half the size of the previous problem; further the sum of the sizes of the new problems is less than the size of the previous problem and hence the processor count is dominated by the first stage. Thus the algorithm takes $O(\log^2 n)$ time using $Q(n)$ processors.

Other NC algorithms for finding a DFST in a DAG are known [GhBh].

In the next two sections we present parallel algorithms to find minimum feedback sets in rfg's. Our algorithms require computation on the head dominator tree T_h of the input rfg

$G=(V,A,r)$. For this we will use a variant of tree contraction. We conclude this section with a description of this modified tree contraction method.

Recall the a chain in a directed graph G is a path $<v_1,\cdots,v_k>$ such that each v_i has exactly one incoming arc and one outgoing arc in G. A *maximal chain* is one that cannot be extended. A *leaf chain* $<v_1,\cdots,v_{l-1},v_l>$ in a rooted tree $T=(V,A,r)$ consists of a maximal chain $<v_1,\cdots,v_{l-1}>$, with v_l the unique child of v_{l-1}, and with v_l, a leaf.

The two tree operations we use in our modified tree contraction method are Rake and Shrink. As before, the Rake operation removes leaves from the tree. The Shrink operation shrinks each maximal leaf chain in the current tree into a single vertex.

Lemma 3.4 In the modified tree contraction method, $O(\log n)$ applications of Rake followed by Shrink, suffice to transform any n node tree into a single vertex.

Proof Consider another modified tree contraction algorithm in which the Shrink operation shrinks all maximal chains, including leaf chains, into a single vertex (one for each chain). This modification certainly requires no more steps than regular tree contraction, and hence by the result in [MiRe], transforms any n node tree into a single vertex in $O(\log n)$ time. But the number of applications of Rake followed by Shrink in the above modified tree contraction method is exactly the same as that in our modified tree contraction method, since the only difference is that a chain gets shrunk in several stages, rather than all at once.[]

In our algorithms for minimum feedback sets, we will associate appropriate computation with the Rake and Shrink operations in order to obtain the desired result.

4. NC Algorithm for Finding a Minimum FVS in an Unweighted Rfg

We first review the basic ideas in Shamir's polynomial time sequential algorithm [Sh]. Given an rfg $G=(V,A,r)$ together with a partial FVS S for G, a head v in G is *active* if there is a DAG path from v to a corresponding tail, which is not cut by vertices in S. A *maximal active head* v is an active head such that none of its proper DAG descendants in G is an active head.

The following theorem is established in [Sh].

Theorem 4.1 [Sh] Let $G=(V,A,r)$ be an rfg, and let S be a subset of a minimum FVS in G. If v is a maximal active head in G with respect to S, then $S \bigcup \{v\}$ is also a subset of a minimum FVS in G.

Using Theorem 4.1, we obtain the following algorithm, based on the head dominator tree, to construct a minimum FVS for an rfg.

4.1 Minimum FVS Algorithm

Input: An rfg $G=(V,A,r)$ together with its head dominator tree T_h.

Output: A set $S \subseteq V$ which is a minimum FVS for G.

1. Initialize $S \leftarrow \phi$.

2. *Repeat*

a) $S \leftarrow S \bigcup L$, where L is the set of leaves in T_h.

b) $G \leftarrow G-L$, $T_h \leftarrow T_h-L$.

c) Find U, the set of heads in current G that are not active in G.

d) For each vertex v not in U, find its closest proper ancestor w in T_h that is not in U, and make w the parent of v; remove all vertices in U from T_h.

until $T_h=\phi$.

We implement the above tree computations using our modified tree contraction method. In order to obtain a processor-efficient implementation of the above algorithm, we define T'_h, *the head-tail dominator tree of* G. For each vertex u in G that is a tail of a back arc but not the head of any back arc in G, we set parent of u in T'_h to be v, where v is a head of a back arc in G, v dominates u in G, and no child of v in T_h dominates u in G. T'_h is the tree obtained from T_h by including these tail vertices of G (all of which will be leaves in T'_h). T'_h can be computed from G in a manner similar to the computation of T_h (Algorithm 3.2). Our parallel algorithm will

perform modified tree contraction on T_h, and will also transform T'_h to keep track of the current structure of G.

The computation associated with a Rake step is exactly one application of step 2 of the above algorithm: we add the leaves of the current tree T_h to S, and delete them from T_h. In T'_h we delete the subtree rooted at each of the vertices we deleted from T_h. We then determine the active heads in the current G. A head h is active in the current G if and only if it lies in T_h, and at least one of its corresponding tails lies in T'_h. This is easily determined for all heads in constant time with $O(m)$ processors on a COMMON PRAM by having a processor for each back arc (u,v), which informs vertex v if u if in T'_h. The set U is then determined as the set of heads that are not currently active. Each vertex in T_h (T'_h) which is not in U finds its closest proper ancestor in T_h (T'_h) that is not in U, and makes it its new parent. Vertices in U are then deleted from both T_h and T'_h. This computation can be done with tree contraction and takes $O(\log n)$ time with $O(n)$ processors on an EREW PRAM.

Now consider the operation Shrink, which shrinks each maximal leaf chain in T_h into a single vertex. During the Shrink operation we will identify the vertices in these leaf chains that belong to S based on the following observation. Let $C=<v_1,\cdots,v_l>$ be a leaf chain, where v_l is the leaf. Suppose v_i is in the minimum FVS S. Then the largest $j<i$ such that v_j is in S (if such a j exists) is immediately determined as the largest $j<i$ for which there is some back edge (u,v_j) such that v_j has a path to u in $G-\{v_i\}$. This is because v_i dominates all $v_k, k>i$, hence any cycle containing $v_j, j<i$, which is cut by a vertex v_k with $k>i$ is certainly cut by v_i; thus v_j will become a maximal active head if v_i is added to S.

Our computation for the Shrink operation determines for each v_i in C, the largest $j<i$ (if it exists) such that v_j is in S if v_i is in S. Then for each i for which j exists, we place a pointer from v_j to v_i. This defines a forest F on $\{v_1,\cdots,v_l\}$. Since v_l is a leaf in T_h it belongs to S. Hence the vertices in C that belong to S are precisely those from which v_l is reachable in F. We identify these vertices using regular tree contraction and add them to S.

We now describe a method to compute v_j for each v_i. Each v_k determines the largest i ($i \geq k$) such that v_i dominates all of the corresponding tails of v_k, and places this value in the kth location of an array $A[1..l]$. Each vertex v_i inspects this array and finds the largest position j ($j < i$) such that $A[j] < i$. Then clearly v_j is the unique vertex in the leaf chain such that v_j is in S if v_i is in S.

Each v_i can determine its corresponding v_j in $O(\log l)$ time with l processors on an EREW PRAM, and hence this computation can be done for all vertices in the leaf chain in $O(\log n)$ time with n^2 processors. (We do not attempt to be more efficient with this computation since the overall algorithm requires $Q(n)$ processors and thus an $O(n^2)$ processor bound is adequate for our needs.)

By Lemma 3.4, $O(\log n)$ applications of Rake and Shrink operations suffice to contract T_h to a single vertex, and at this point we will have constructed a minimum FVS S. Thus this gives a parallel algorithm to find a minimum FVS in an rfg in $O(\log^2 n)$ time using $Q(n)$ processors. A high-level description of the parallel algorithm is given below.

4.2 Parallel Minimum FVS Algorithm

Input: An rfg $G=(V,A,r)$.

Output: A set $S \subseteq V$ which is a minimum FVS for G.

1. Form T_h and T'_h; initialize $S \leftarrow \phi$.

2. *Repeat*

a) Rake leaves

i) Place leaves of T_h in S.

ii) In T'_h, delete the subtree rooted at each vertex raked in step i) in T_h.

iii) For each vertex v in T_h, determine if v is active by checking if any one of its corresponding tails is in T'_h.

iv) Let U be the set of vertices in T_h that are not active heads. For each vertex v in T_h (T'_h)

that is not in U, find its closest proper ancestor w that is not in U and make it its new parent in T_h (T'_h).

v) Delete vertices in U from T_h and T'_h.

b) Shrink leaf chains

Let $<v_1,v_2,\cdots,v_l>$ be the leaf chain, with v_l the leaf.

i) For each v_k find the largest i ($i \geq k$) such that v_i dominates all of the corresponding tails of v_k, and place this value in location k of array $A[1..k]$.

ii) For each v_i use array $A[1..k]$ to find the largest position j ($j<i$) such that $A[j]<i$. Place an arc from v_j to v_i in an auxiliary graph F on vertices $v_1,\cdots,v_l$.

iii) Find the set of vertices from which v_l is reachable in F and add these vertices to S.

iv) Delete vertices $v_1,\cdots,v_l$ from T_h and delete the subtrees rooted at these vertices from T'_h.

v) Perform parts iii, iv and v of the Rake step.

until $T_h=\phi$.

5. Finding a Minimum FAS in an Unweighted Rfg and Related Problems

We first state some definitions and results from [Ra1], which gives a polynomial-time sequential algorithm for finding a minimum FAS in an rfg. We then give an RNC algorithm for this problem and related results.

A *flow network* $G=(V,A,s,t,C)$ is an arc-weighted directed graph with vertex set V and arc set A, where s and t are vertices in V called the *source* and *sink* respectively, and C is the *capacity function* on the arcs which specifies the arc weights, which are always nonnegative. The *maximum flow* problem asks for a *flow* of maximum value from s to t (see [Ev, FoFu, PaSt, Ta2] for definition of a flow). A *cut* C separating s and t is a set of arcs that breaks all paths from s to t. The *capacity* of C is the sum of the capacities of arcs in C. A *minimum cut* separating s and t is a cut of minimum capacity. It is well-known that the value of a maximum flow is equal

to the value of a minimum cut [FoFu].

Let $G=(V,A,r)$ be an arc-weighted reducible graph and let v be the head of a back arc in G. Let $b_1=(u_1,v_1),\cdots,b_r=(u_r,v_r)$ be the back arcs in G whose heads are dominated by v. The *dominated back arc vertex set of* v is the set $V_v=\{v'\in V \mid v'$ lies on a DAG path from v to some $u_i, i=1,\cdots,r\}$. It is easy to see that v dominates all vertices in V_v.

Definition 5.1 Let $G=(V,A,r)$ be an arc-weighted reducible graph with nonnegative arc weights, and let v be the head of a back arc in G. For convenience of notation we denote $G_s(V_v)$, the subgraph of G induced by the dominated back arc vertex set of v, by $G_s(v)$. The *maximum flow network of* G *with respect to head* v is a flow network $G_m(v)$ formed by splitting each head h in $G_s(v)$ into h and h' (see figure 1). All DAG arcs entering or leaving the original head h will enter or leave the newly formed h; all back arcs entering the original h will enter h'. There will be an arc of infinite capacity from h' to a new vertex t. All other arcs will inherit their capacities from their weights in G. We will interpret v as the source and t as the sink of $G_m(v)$.

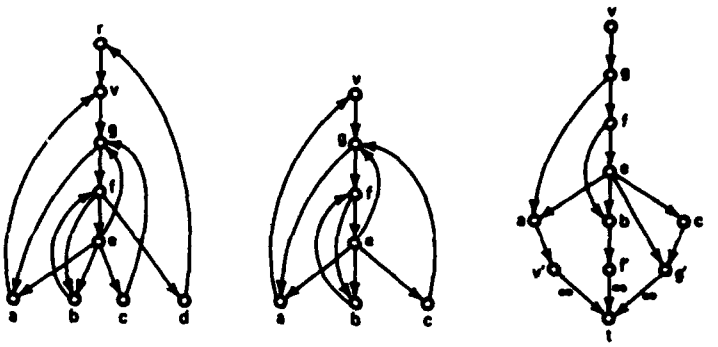

figure 1
Constructing $G_m(v)$ from $G=(V,A,r)$

Definition 5.2 Let $G=(V,A,r)$ be an arc-weighted reducible graph. We define $G_{mm}(v)$, *the min-*

cost maximum flow network with respect to head v inductively as follows:

a. If v dominates no other head in G then $G_{mm}(v)=G_m(v)$.

b. Let $v_1, \cdots, v_r$ be the heads immediately dominated by v in G and let the capacity of minimum cut in $G_{mm}(v_i)$ be $c_i, i=1, \cdots, r$. Then $G_{mm}(v)=(V, A)$ where V is the same as the vertex set for $G_m(v)$ and A = {arcs in $G_m(v)$} $\cup$ {arcs in $G_{mm}(v_i), i=1, \cdots, r$} $\cup F_v$, where F_v = $\{f_{vv_i}=(v, v_i) \mid i=1, \cdots, r$, with capacity of f_{vv_i} equal to $c_i\}$.

We call F_v the *mincost-arc set for head* v; if j is a head immediately dominated by head i then f_{ij} is the *mincost arc from head* i *to head* j.

Figure 2 gives an example of $G_{mm}(v)$.

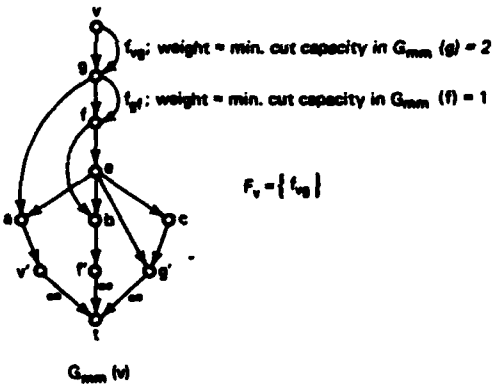

figure 2
The mincost maximum flow network with respect to head v for the graph $G=(V, A, r)$ of figure 1

It is established in [Ra1] that following algorithm determines the cost of a minimum FAS in an arc-weighted reducible graph G. If G is an unweighted graph, then by the result in [Ra4] this value also gives the maximum number of arc disjoint cycles in G.

5.1 Minimum FAS Algorithm for Reducible Flow Graphs

Input: A reducible graph $G=(V,A,r)$ with nonnegative weights on arcs.

Output: The cost of a minimum FAS for G.

begin

1. Preprocess G: Label the heads of back arcs in G in postorder. Derive the head dominator tree T_h for G. Introduce a pointer from each vertex i in T_h (except r) to its parent h_i. Let the number of heads be h.

2. For $i=1,\cdots,h$ process head i:

   a. Find the capacity of minimum cut, c_i, in $G_m(i)$.

   b. If $i\neq h$ then introduce an arc of weight c_i from h_i to i in G. (Note that G changes during the execution of the algorithm so that $G_m(i)$ is the same as $G_{mm}(i)$ if G were unchanged.)

3. Output c_h as cost of minimum FAS for G.

end.

We implement the above tree computations once again using our modified tree contraction method. For a Rake step, we form $G_m(l)$, for each leaf l in T_h and compute c_l, the capacity of a minimum cut in $G_m(l)$. If $l\neq h$ then we place an arc of capacity c_l from h_l to l. Finally we delete all leaves from the current T_h.

The complexity of the Rake step is dominated by the complexity of computing minimum cuts in the mincost maximum flow networks associated with the leaves of T_h. The total size of all of these networks is $O(m+n)$, where m and n are the number of arcs and vertices, respectively, in G. Hence, using the algorithm in [MuVaVa] to compute minimum cuts, we can perform the Rake step by a randomized algorithm that runs in $O(\log^2 n)$ parallel time with $O(m\cdot n^{5.5})$ processors on an EREW PRAM.

The Shrink operation is a little more involved. We assume some familiarity with the results in [Ra1]. Let $C=<v_1,\cdots,v_l>$ be a leaf chain in T_h, with v_l the leaf. During the Shrink operation, we will determine the capacity of a minimum cut in $G_{mm}(v_i), i=1,\cdots,l$. We can then use this to construct to $G_{mm}(v_1)$, and hence the current G.

Our parallel algorithm will process $G_{mm}(v_1)$ in chunks. For this we develop some notation. Let $G^{i,j}$, $1 \le i \le j \le l$ be the graph $G_{mm}(v_i)$ with

a) All arcs dominated by v_j deleted and replaced by single arc of infinite capacity from v_j to t,

b) The capacities of the mincost arcs $(v_{k-1},v_k), k=i+1, \cdots, j-1$ set to ∞, and

c) The mincost arc (v_{j-1},v_j) deleted.

The graph $G^{i,j+1}$ will denote $G_{mm}(v_i)$ with the capacities of all mincost arcs $(v_{k-1},v_k), k=i+1, \cdots, j$ set to ∞. We will denote the mincost arc from v_{j-1} to v_j, $j>i$ in $G_{mm}(v_i)$ by e_j and call it a *chain mincost arc*.

We will use the notation $n^{i,j}$ to denote the value of a minimum cut in $G^{i,j}$, and m_i to denote the value of a minimum cut in $G_{mm}(v_i)$. We define $n^{i,j}=0$ if $i=j$ and $m_i=0$ if $i=l+1$.

Lemma 5.1 Let M be a minimum cut in $G_{mm}(v_i)$ that contains a chain mincost arc e_j. Then v_{j-1} is separated from v_j by M.

Proof Consider the vertex partition $S \bigcup T$ induced by M, where S is the set of vertices in the component containing v_i in $G-M$. If both v_{j-1} and v_j are in S (or T) then we can remove e_j from M and still have a cut, contradicting the fact that M is a minimum cut. If v_{j-1} is in S and v_j is in T, then v_{j-1} is separated from v_j by the cut as required. Finally, v_j in S and v_{j-1} in T is not possible since every path from v_i to v_j must pass through v_{j-1}.[]

Lemma 5.2 Let M be a minimum cut in $G_{mm}(v_i)$. Then M contains at most one chain mincost arc.

Proof Suppose M contains two chain mincost arcs e_j and $e_k, j<k$. By Lemma 1, v_{j-1} is separated from v_j by M. Hence every path from v_i to v_k is cut by $M-\{e_k\}$. Thus $M-\{e_k\}$ is a cut for $G_{mm}(v_i)$ contradicting the fact that M is a minimum cut.[]

Lemma 5.3 Let M be a minimum cut in $G_{mm}(v_k)$, for some k, $1 \le k \le l$, and let M separate v_i from v_{i-1} for some $i>k$. Let N be a minimum cut in $G^{k,i}$. Then $N \bigcup \{e_i\}$ is a minimum cut for $G_{mm}(v_k)$, $n^{k,i}+m_i$ is the value of a minimum cut in $G_{mm}(v_k)$, and $N \bigcup M_i$ is a minimum FAS for $G_s(v_k)$, where M_i is a minimum FAS for $G_s(v_i)$.

Proof Let $G'_{mm}(v_k)$ be $G_{mm}(v_k)$ with the capacities of mincost arcs $(v_{j-1},v_j), j=k+1,k+2,\cdots,i-1$ increased to ∞. M will continue to be a minimum cut in $G'_{mm}(v_k)$ since M separates v_i from v_{i-1} in $G_{mm}(v_k)$. Since M is a minimum cut, none of the arcs in M are dominated by v_i, since M would be a cut even if all such arcs are deleted from it. Since M must contain e_i, we can write $M=M' \bigcup \{e_i\}$, where M' contains no mincost arc.

Now consider N. N is a minimum cut in $G_{\infty}^{k,i}$. Since all of the mincost arcs in $G_{\infty}^{k,i}$ have infinite capacity, N contains no mincost arc, and every arc in N appears in $G_{mm}(v_k)$. N must separate v_{i-1} from v_i in $G_{\infty}^{k,i}$, since there is a path of infinite capacity from v_k to v_{i-1} and from v_i to t in $G_{\infty}^{k,i}$. Hence $N \bigcup \{e_i\}$ separates v_i from t in $G_{mm}(v_k)$. Finally, $|N| \leq |M'|$ since M' is a cut for $G_{\infty}^{k,i}$. Hence $N \bigcup \{e_i\}$ is a minimum cut for $G_{mm}(v_k)$.

By definition, the capacity of e_i is the capacity of a minimum cut in $G_{mm}(v_i)$, which is m_i. Hence $n^{i,k}+m_i$ is the capacity of minimum cut in $G_{mm}(v_k)$.

Finally, since N has no mincost arcs and $N \bigcup \{e_i\}$ is a minimum cut for $G_{mm}(v_k)$, it follows from the results in [Ra1] that $N \bigcup M_i$ is a minimum FAS for $G_s(v_k)$, where M_i is any minimum FAS for $G_s(v_i)$.[]

Lemma 5.4 Let $1 \leq l=i_1<i_2<\cdots<i_r=j \leq l+1$ be any sequence of indices. Let N_k be a minimum cut in $G_{\infty}^{i_k,i_{k+1}}$, $k=1,\cdots,r-1$. Then $N_1 \bigcup N_2 \bigcup \cdots \bigcup N_{r-1} \bigcup M_j$ is an FAS for $G_s(v_i)$, where M_j is an FAS for $G_s(v_j)$.

Proof By induction on r.

Base: $r=2$. Suppose $j \leq l$. N_1 is a minimum cut in $G_{\infty}^{i,j}$. Hence N_1 must separate v_{j-1} from v_j in $G_{\infty}^{i,j}$, since there is a path of infinite capacity from v_i to v_{j-1} and from v_j to t in $G_{\infty}^{i,j}$. Hence $N_1 \bigcup \{e_j\}$ is a cut in $G_{mm}(v_i)$. Hence by the results in [Ra1], $N_1 \bigcup M_j$ is an FAS for $G_s(v_i)$, where M_j is any FAS for $G_s(v_j)$. If $j=l+1$ then we note that any cut in $G_{\infty}^{i,l+1}$ is an FAS for $G_s(v_i)$ as well.

Induction step: Assume that the statement of the lemma is true for all sequences of indices of length $r=r'-1$ or less, and let $r=r'$. By the base case, $N_1 \bigcup M_{i_2}$ is an FAS for $G_s(v_i)$, where M_{i_2} is

any FAS for $G_s(v_{i_2})$. By the inductive hypothesis, $N_2 \cup \cdots \cup N_r \cup M_j$ is an FAS for $G_s(v_{i_2})$. Hence $N_1 \cup \cdots \cup N_r \cup M_j$ is an FAS for $G_s(v_i)$.[]

Lemma 5.5 Let $1 \le i = i_1 < i_2 < \cdots < i_r = j \le l+1$ be a sequence of indices such that there exists a minimum cut in $G_{mm}(v_{i_k})$ that separates $v_{i_{k+1}-1}$ from $v_{i_{k+1}}, k=1, \cdots, p$, where $p=r-1$ if $j \le l$ and $p=r-2$ if $j=l+1$. Let N_k be a minimum cut in $G_{\infty}^{i_k, i_{k+1}}, k=1, \cdots, r-1$. Let M_j be a minimum FAS for $G_s(v_j)$. Then $N_1 \cup \cdots \cup N_{r-1} \cup M_j$ is a minimum FAS for $G_s(v_i)$.

Proof By induction on r.

Base: $r=2$. Then $i=i_1$ and $j=i_2$. If $j \le l$ then the result follows from Lemma 5.3. If $j=l+1$ then no minimum cut in $G_{mm}(v_i)$ separates any v_k from v_{k-1}, $i < k \le l$. Hence any minimum cut in $G_{\infty}^{i,l+1}$ is a minimum cut for $G_{mm}(v_i)$ and the result follows.

Induction step: Assume that the statement of the lemma is true for all sequences of indices of length $r=r'-1$ or less, and let $r=r'$. By Lemma 5.3 $N_1 \cup M_{i_2}$ is a minimum FAS for $G_s(v_i)$, where M_{i_2} is any minimum FAS for $G_s(v_{i_2})$. By the inductive hypothesis, $N_2 \cup \cdots \cup N_r \cup M_j$ is a minimum FAS for $G_s(v_{i_2})$. Hence $N_1 \cup \cdots \cup N_r \cup M_j$ is a minimum FAS for $G_s(v_i)$.[]

Let $l^{i,j}$ be the minimum value of the sum $n^{i,i_2}+n^{i_2,i_3}+\cdots+n^{i_{r-1},j}$, where $1 \le i < i_2 < \cdots < i_{r-1} < j \le l+1$, and the indices i_j and their number $r-2 \ge 0$ are allowed to range over all permissible values. Note that $l^{i,j+1}=n^{i,j+1}$.

Lemma 5.6 Let $1 \le i = i_1 < i_2 < \cdots < i_r = j \le l+1$ be a sequence of indices such that there exists a minimum cut in $G_{mm}(v_{i_k})$ that separates $v_{i_{k+1}-1}$ from $v_{i_{k+1}}, k=1, \cdots, p$, where $p=r-1$ if $j \le l$ and $p=r-2$ if $j=l+1$. Then the cost of a minimum FAS for $G_s(v_i)$ is $l^{i,j}+m_j$.

Proof By Lemma 5.4, there exists an FAS in $G_s(v_i)$ with cost $n^{i,j_2}+n^{j_2,j_3}+\cdots+n^{j_{s-1},j}+m_j$ for any sequence of indices $i < j_2 < \cdots < j_{s-1} < j$. By Lemma 5.5, a minimum FAS in $G_s(v_i)$ has cost $n^{i,i_2}+n^{i_2,i_3}+\cdots+n^{i_{r-1},j}+m_j$. Hence the indices $i_2, \cdots, i_{r-1}$ will contribute to the minimum in the expression for $l^{i,j}$. Thus the cost of a minimum FAS for $G_s(v_i)$ is $l^{i,j}+m_j$.[]

Corollary $m_i=l^{i,i+1}$.

Lemma 5.7 Let $h=\lceil (i+j)/2 \rceil$. Then $l^{i,j}=\min_{i\le x\le h<y\le j} l^{i,x}+n^{x,y}+l^{y,j}$.

Proof An easy proof shows that $l^{i,j}\le \min_{i\le x\le h<y\le j} l^{i,x}+n^{x,y}+l^{y,j}$ and $\min_{i\le x\le h<y\le j} l^{i,x}+n^{x,y}+l^{y,j}\le l^{i,j}$.[]

The characterization in Lemma 5.7 leads to the following implementation of the Shrink step in the parallel algorithm to find the value of a minimum FAS in an rfg.

5.2 Parallel FAS Algorithm for SHRINK Step

Input Graph $G_{mm}(v_1)$ with its associated head dominator path $<v_1,v_2,\cdots,v_l>$.

1. For $i=1,\cdots,l$ do

   Compute $l^{i,i+1}$ by finding the value of a minimum cut in $G^{i,i+1}_{st}$.

2. For $i=1,\cdots,l$

   For $j=i+1,\cdots,l+1$ do

      Compute $n^{i,j}$ by finding the value of a minimum cut in $G^{i,j}_{st}$.

3. For $k=1,2,\cdots,\lceil \log l \rceil$ do

   for $i=1,\cdots,l-2^k$

      for $j=i+2^{k-1}+1, i+2^{k-1}+2,\cdots,i+2^k$ do

         Let $h=\lceil (i+j)/2 \rceil$

         $l^{i,j}=\min_{i\le p\le h<q\le j}(l^{i,p}+n^{p,q}+l^{q,j})$

4. For $i=1,\cdots,l$ output $l^{i,i+1}$ as value of minimum cut for $G_{mm}(v_i)$.

Let $G_{mm}(v_1)$ have r vertices and s arcs. Step 1 requires the computation of $l\le r$ minimum cuts in parallel on graphs whose total size is $O(r+s)$. Hence this step can be executed by a randomized algorithm in $O(\log^2 r)$ time with $O(s\cdot r^{3.5})$ processors on an EREW PRAM, using the algorithm in [MuVaVa]. Step 2 requires the computation of $O(l^2)$ minimum cuts and in the worse case this requires $O(\log^2 r)$ time using $O(s\cdot r^{5.5})$ processors for a randomized algorithm on an EREW PRAM. The inner loop of step 3 (using indices i and j) can be executed in constant

time with $O(l^2)=O(r^2)$ processors and hence step 3 can be executed in $O(\log r)$ time with $O(r^2)$ processors by a deterministic algorithm on an EREW PRAM. Thus the complexity of the Shrink step is dominated by step 2.

The FAS algorithm uses the above Rake and Shrink operations $\log n$ times. Hence it is a randomized algorithm that runs on an EREW PRAM in $O(\log^3 n)$ time using $O(m \cdot n^{5.5})$ processors. This is an RNC algorithm.

At this point we have the value of a minimum cut in all $G_{mm}(v)$, where v is the head of a back arc in G. Hence we can construct $G_{mm}(v)$ for each such v and find a minimum cut in each of these graphs in parallel using the RNC algorithm of [MuVaVa]. From this we can extract a minimum FAS for G as follows: Place a pointer from each mincost arc (u,v) in any of these minimum cuts to the minimum cut for $G_{mm}(v)$. Now a minimum FAS for G consists of the set of arcs in G that are in some minimum cut that is reachable from the minimum cut for $G_{mm}(r)$ in this pointer structure. This is an easy NC computation and thus we obtain an RNC algorithm to find a minimum FAS in the rfg G.

Finally we present some results on the parallel complexity of finding feedback sets in weighted rfg's.

Lemma 5.8 The following problems are reducible to one another through NC reductions.

1) Finding a minimum FAS in an unweighted rfg.

2) Finding a minimum weight FAS in an rfg with unary weights on arcs.

3) Finding a minimum weight FVS in an rfg with unary weights on vertices.

4) Finding a minimum cut in a flow network with capacities in unary.

Proof: Polynomial-time reductions between between 1), 2), and 3) are given in [Ra1]. We note that all of these reductions are NC reductions as well. We show that 4) reduces to 2): We use the NC reduction in [Ra2] from the problem of finding a minimum cut in a general flow network G to the problem of finding a minimum cut in an acyclic flow network N. Minimum cut for N can be obtained by finding a minimum weight FAS in graph G' derived from N by coalescing source

and sink.

We also have the result that 2) reduces to 4) since our parallel FAS algorithm uses $O(\log n)$ applications of an algorithm for 4) together with some additional NC computation.[]

Lemma 5.9 The following two problems are P-complete:

1) Finding a minimum weight FAS in an rfg with arbitrary weights on arcs.

2) Finding a minimum weight FVS in an rfg with arbitrary weights on vertices.

Proof: It is established in [Ra2] that finding minimum cut in acyclic networks is P-complete. Let G be an acyclic network with source s and sink t. Let G' be formed from G by combining s and t into a single vertex r. Then G' is an arc-weighted rfg rooted at r and a minimum weight FAS in G gives a minimum cut in G. Hence part 1 of the lemma follows.

We can reduce the minimum weight FAS problem on rfg's to the minimum weight FVS problem on rfg's by replacing each arc in the arc-weighted graph G by two arcs (u,w) and (w,v) and assigning to w the weight of arc (u,v). The original vertices in G are assigned a weight $n \cdot W$, where W is the maximum weight of any arc in G, and n is the number of vertices in G. It is easy to see that a minimum-weight FVS in the new graph gives back a minimum weight FAS in G having the same weight. This establishes part 2 of the lemma.[]

References

[AhUl] A. V. Aho, J. D. Ullman, *Principles of Compiler Design*, Addison Wesley, 1977.

[Al] F. E. Allen, "Program optimization," *Annual Rev. Automatic Prog.*, vol. 5, Pergamon Press, 1969.

[BoDAPe] D. P. Bovet, S. De Agostino, R. Petreschi, "Parallelism and the feedback vertex set problem," *Inform. Proc. Lett.*, vol. 28, 1988, pp. 81-85.

[Fl] R. W. Floyd, "Assigning meanings to programs," *Proc. Symp. Appl. Math.*, 19, pp. 19-32, 1967.

[FoFu] L. R. Ford, Jr., D. R. Fulkerson, *Flows in Networks*, Princeton Univ. Press, Princeton, NJ, 1962.

[GaMi] H. Gazit, G. L. Miller, "An improved parallel algorithm that computes the bfs numbering of a directed graph," *Inform. Proc. Lett.*, vol. 28, 1988, pp. 61-65.

[GaTa] H. Gabow, R. E. Tarjan, "A linear-time algorithm for a special case of disjoint set union," *Proc. 15th Ann. ACM Symp. on Theory of Computing*, 1983, pp. 246-251.

[GhBh] R. K. Ghosh, G. P. Bhattacharya, "Parallel search algorithm for directed acyclic graphs," BIT, 1982.

[GoTa] A. Goldberg, R. E. Tarjan, "A new approach to the maximum flow problem," *Proc. 18th Ann. ACM Symp. on Theory of Computing*, 1986, pp. 136-146.

[Ha] D. Harel, "A linear algorithm for finding dominators in flow graphs and related problems," *17th Ann. Symp. on Theory of Computing*, 1985, pp. 185-194.

[HeUl] M. S Hecht, J. D. Ullman, "Characterization of reducible flow graphs," *JACM*, 21.3, 1974, pp. 167-175.

[KarRa] R. M. Karp, V. Ramachandran, "Parallel algorithms for shared-memory machines," in *Handbook of Theoretical Computer Science*, J. Van Leeuwan, ed., North Holland, 1989, to appear.

[KaUpWi] R. M. Karp, E. Upfal, A. Wigderson, "Constructing a perfect matching is in Random NC," *Proc. 17th ACM Ann. Symp. on Theory of Computing*, 1985, pp. 22-32.

[MiRe] G. L. Miller, J. H. Reif, "Parallel tree contraction and its application," *Proc. 26th Ann. IEEE Symp. on Foundations of Comp. Sci.*, 1985, pp. 478-489.

[MuVaVa] K. Mulmuley, U. Vazirani, V. Vazirani, "Matching is as easy as matrix inversion," *Proc. 19th Ann. ACM Symp. on Theory of Computing*, 1987, pp. 345-354.

[PaSt] C. H. Papadimitriou, K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall, Englewood Cliffs, NJ, 1982.

[PaGoRa] S. R. Pawagi, P. S. Gopalakrishnan, I. V. Ramakrishnan, "Computing dominators in parallel," *Inform. Proc. Lett.*, vol. 24, 1987, pp. 217-221.

[Ra1] V. Ramachandran, "Finding a minimum feedback arc set in reducible graphs," *Journal of Algorithms*, vol. 9, 1988, pp. 299-313.

[Ra2] V. Ramachandran, "The complexity of minimum cut and maximum flow problems in an acyclic network," *Networks*, vol. 17, 1987, pp. 387-392.

[Ra3] V. Ramachandran, "Fast parallel algorithms for reducible flow graphs," in *Concurrent Computations: Algorithms, Architecture and Technology*, S. K. Tewksbury, B. W. Dickinson and S. C. Schwartz, ed., Plenum Press, New York, NY, 1988, pp. 117-138.

[Ra4] V. Ramachandran, "A minimax arc theorem for reducible flow graphs," tech. report 87-001, International Computer Science Institute, Berkeley, CA, 1987.

[Sh] A. Shamir,"A linear time algorithm for finding minimum cutsets in reducible graphs," *SIAM J. Computing*, vol. 8, 1979, pp. 645-655.

[Ta1] R. E. Tarjan, "Testing flow graph reducibility," *JCSS*, vol. 9, 1974, pp. 355-365.

[Ta2] R. E. Tarjan, *Data Structures and Network Algorithms*, SIAM, Philadelphia, PA, 1983.